6.005 Collaborative Editor Design Document

angelaz-mturek-nedmonds

Table of Contents

Purpose

To design a collaborative editor that allows multiple users to work on a single document simultaneously across a network.

Introduction

The collaborative editor contains 2 major components: the server and the client. Please see the **Diagram** Section below for the System Diagram and the State Machine.

The server is made up of SEditServer, SEditThread, and SEditDocument, the functionalities of each will be described in detail in the **Server** section of the Design Document. The server serves as a central unit for all the clients to connect to, listens for and processes client requests, and maintains the edit history for all the documents.

The client is made up of the Model, View, Controller components, each will be described in detail in the **Client** section of the Design Document. Each client can connect and disconnect to the server, open existing documents that the server contains, create new documents, rename these documents, and concurrently edit the documents.

The server and clients communicate with each other via a Server-Client communication protocol, whose grammar is defined in the **Messages** section.

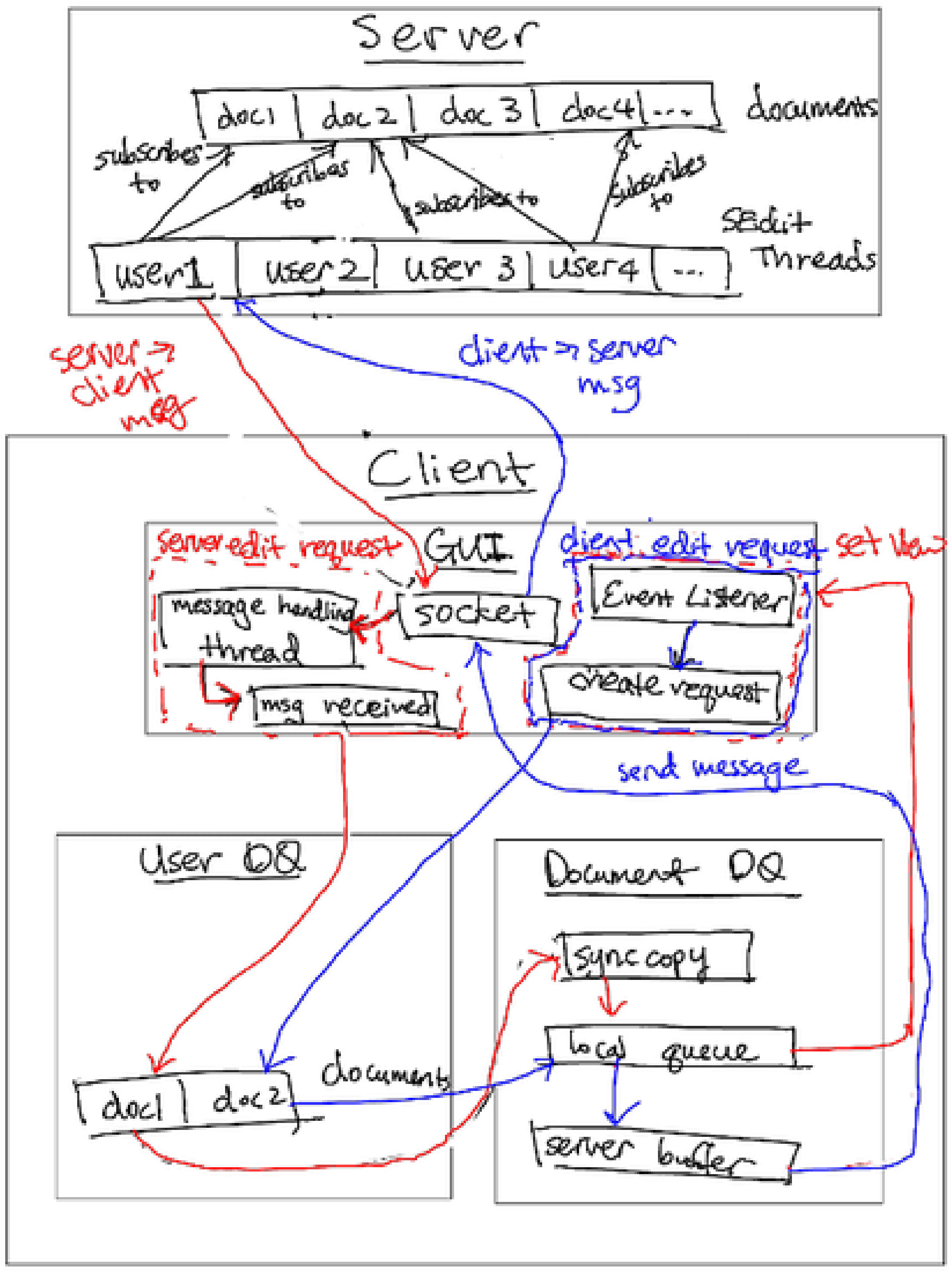
In addition, all components of the editor are designed to be thread-safe and free of concurrency bugs. This is accomplished by employing techniques such as confinement, immutability, using threadsafe datatypes, and explicit synchronization, as described in the **Thread Safety** section.

Each component is extensively tested individually and collectively. Following the test-first programming principle, we wrote test cases for the units Server, GUI, Model, and finally, System. For more information, please see the **Testing** section.

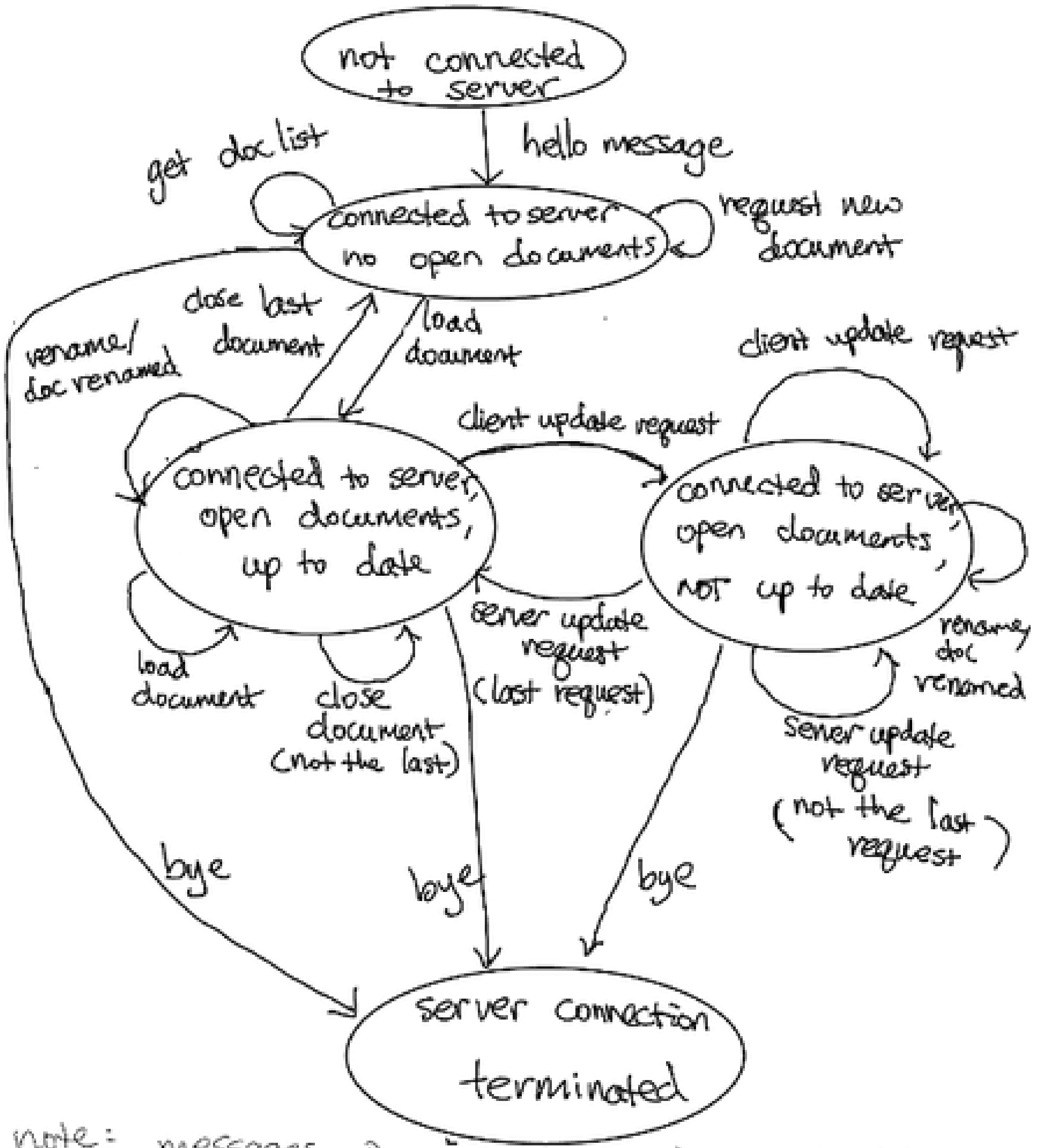
The server can be started by running the `SEditServer.java` file, and the client can be started by running the `Editor.java` file.

Diagrams

System Diagram



Protocol & State Machine


not connected to server
hello message
get doc list
connected to server no open documents
request new document
close last document
load document
rename/ doc renamed
client update request
client update request
connected to server, open documents, up to date
connected to server, open documents, NOT up to date
server update request (last request)
load document
close document (not the last)
rename/ doc renamed
server update request (not the last request)
bye
bye
bye
server connection terminated

note: messages 1) get doc list, 2) request new document, 3) rename / doc renamed do not affect the state, and are implied in the diagram above.

Server

The role of the server is very limited in this design since it does not directly store a copy of the document nor does it apply to it the changes received from individual users. The main point of the server is to hold an instance of a Map called documents. Documents maps a collection of document IDs to SEditDocuments, which are the server representations of the documents. The SEditDocuments store an arraylist serverHistory that accumulates all the changes applied to the document since version 0 (which is by default an empty document). The position of a ServerRequestDQ (the grammar of which is specified in Appendix C) in the arraylist is the number of the version that applying that particular request will produce (it is also attached to the message as its last segment).

When a client connects to the server, the server assigns him a unique user ID in the hello message and then starts listening to incoming requests from the client. Once a change request to a document is received from the user, the server needs to make sure that the indices of that change match the latest version of that document handled by the server. Since we know which version the client was holding when he made that change and we know that all the local changes made by the client in his localQueue prior to this one must have been already been processed by the server, we simply update the current request by applying all the requests in the document's serverHistory that other users made since the version held by the client. A concrete example:

- Two users both hold a document in version 5 with content ABCD
- User 1 inserts an X after the A → "INSERT|1|0|X" with version ID 5
- User 2 inserts a Y after the C → "INSERT|3|0|Y" with version ID 5
- The request sent by User 1 gets processed first and it is just included as it is since the version ID matches the version held by the server
- The request sent by User 2 gets processed second and the server sees that since User 2 sent it not knowing User 1 inserted an X in the text. In order to synchronize the insertion index with the current version (6), the INSERT|1|0|X request is "applied" to the "INSERT|3|0|Y request, updating it to "INSERT|4|0|Y. The updated request is then added to serverHistory and distributed to all the connected clients.

Client

The client consists of two parts, the **Model**, represented by the `UserDQ` class which manages the local version of the document and the **GUI** which groups the View and the Controller.

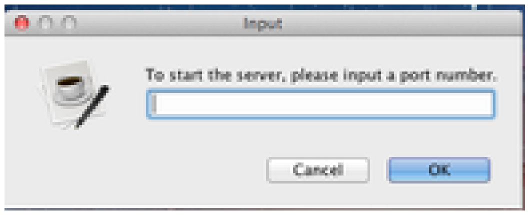
The EditorGUI class manages the connection of a user to the server and displays the current version of the document provided by the model. The GUI responds to 2 kinds of events:

1. The user directly modifies the content of his JTextPane. We do not want this to affect the local version of the document, however, since this would result in data inconsistencies over time and a lot of effort to ensure proper synchronization. Instead, the model places the change into its own `localQueue` which is an ArrayList containing a sequence of changes to be applied to the latest synchronized copy held by the client. The synchronized local copy (`syncCopy`) may only be modified by the server, never by a direct change in the user interface. The client then sends this change to the server to be processed and distributed to other users.
2. A change request is received from the server in a background thread, if this change is coming from this user, it gets removed from the `localQueue` (as it has been processed and is part of the current `syncCopy`). If the change is coming from another user, it is applied to the `syncCopy` (thus increasing its version number) and also to all the requests in `localQueue` (since those are only meaningful relative to the `syncCopy` version which has changes).

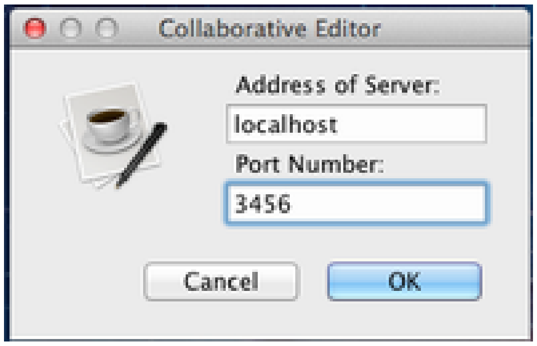
Once all the events have been processed, the GUI updates what the user sees by asking the model for the current `viewCopy` – UserDQ then takes the syncCopy and applies all the changes stacked in `localQueue` to get what the user is supposed to be seeing.

GUI Screenshots:

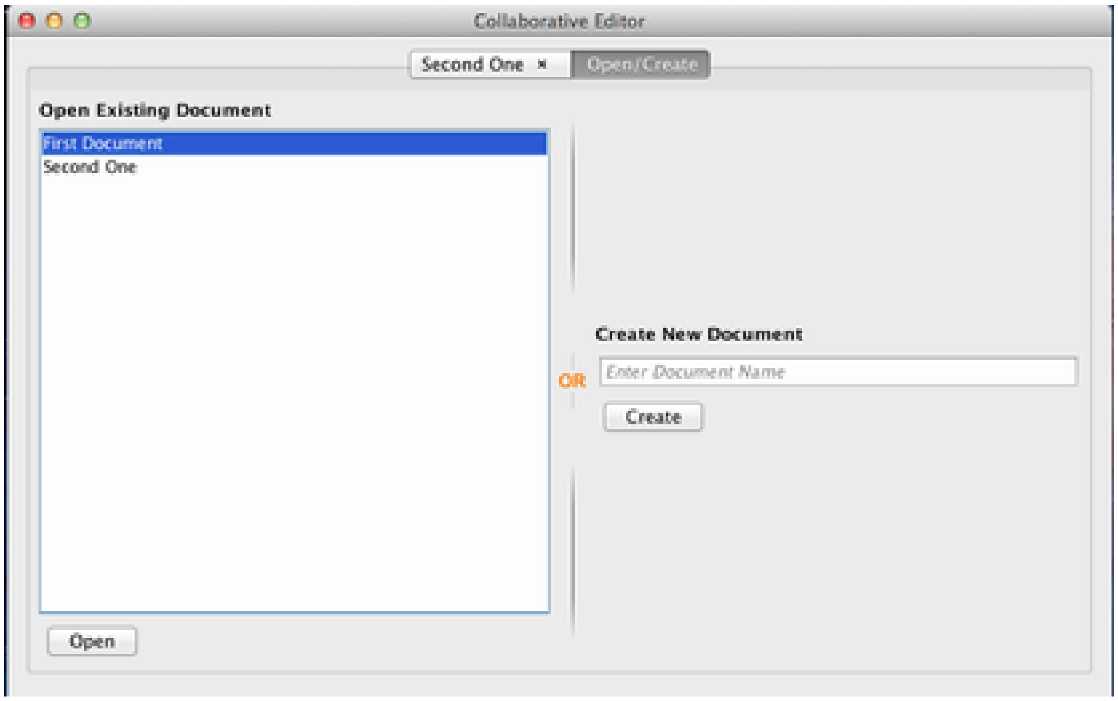
Server Input Port Number Dialog:



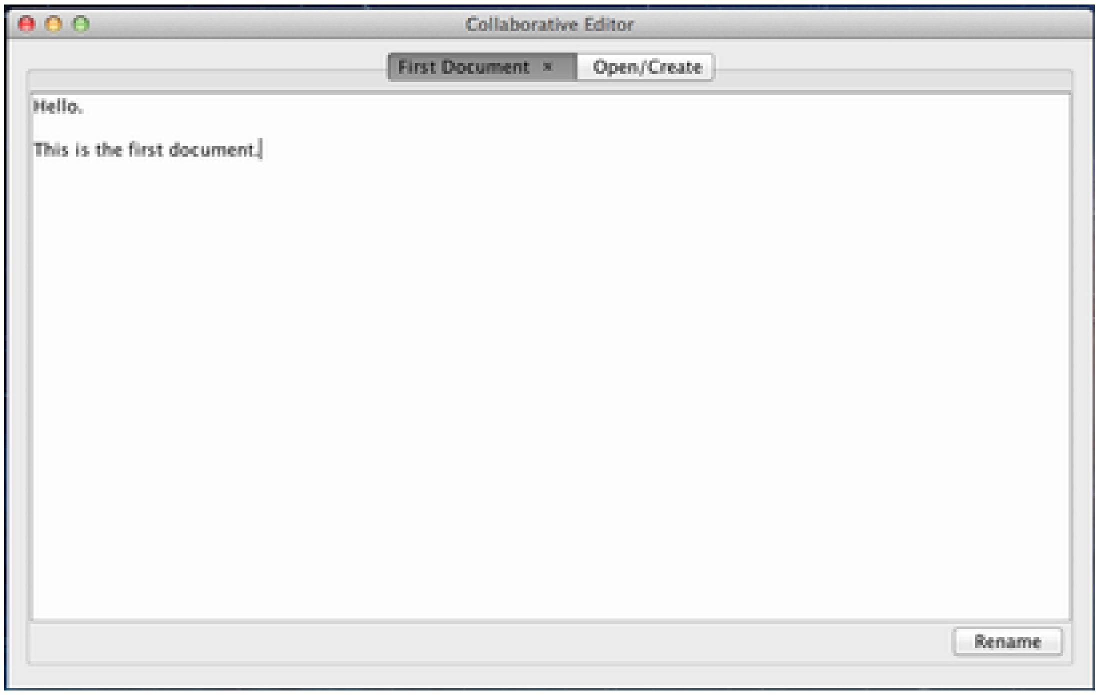
Client Input Server IP & Port Number Dialog:



Client Create New/Open Existing Panel:



Document Editing Pane:



Hello.

This is the first document.

Testing

Testing was done on two levels. Each component (GUI, Server, Model) was tested individually, and then the System was tested as a whole. For the model and server, automatic JUnit tests were written to cover every behavior and test that errors were thrown at the appropriate times. For the GUI and for the System tests were written as a series of instructions with actions and expected results. For details beyond what is below, refer to **Appedix D.**

Details on Model Testing:

Since the actual delays and lags in server-client communication on local network even the Internet are extremely unpredictable and depend on many different inputs we cannot affect, it is not ideal to use the same interface for testing as we are in the actual implementation. For testing, we decided to circumvent all network communication and implement a server simulator that simulates and drives all the events that happen when one actually runs the program but in a much more organized and controlled manner. Events are matched to function calls in the following manner:

UserDQ.createRequest(...) -> simulates a user making a local change in his GUI
UserDQ.pullRequest() -> simulates request sent from user
UserDQ.pushRequest() -> simulates message from server received by user

Server processing is not done automatically since the server is tested separately in SEditTest. Here, server processing of messagesis done manually.

In real use, the difference between pullRequest and the server processing (and in turn between server processing and pushRequest), for example, is determined by the lag of the network. Here, we can time and order them as we desire to simulate concurrent changes and lagging updates.

We also made heavy use of the routers in Number 6 broken by last week's power outage which allowed us to test our system in a real-world environment with substantial lagging. All out tests still passed.

The point of the test cases described is to push the concurrency capabilities of the implementation to the limits with many changes happening at the same time without knowing about other changes made by other users.

Details on System Testing:

System tests are slightly less formal than other tests. While some are quite specific, a portion are simple "stress tests" where the users do everything in their power to break the system.

Messages

There are several kinds of messages that get passed between the server and the client at different points in the execution of the program. Following a successful connection to the server, a client receives a **hello** message that assigns him his UserID. The client then requests all the changes made to the document it is opening by sending a **load** message to get updated to the current server version. During most of the time, all communication between the client and server will be done through **ServerRequestDQ** messages that specify changes to be applied to the document → a client sends one when a changes is made in a GUI and the server distributes them to all the other clients. Requests that are not directly related to changing the content of a document are grouped as **control** messages (load is one of them) and get processed in the GUI. ServerRequestDQ must be forwarded to the model where they are processed.

Unlike in our previous design, we decided that for simplicity all edits should be applied as block changes (since single-character insertion and deletion are just block edits of length 1) and that the proposed system of anchors is not necessary in a String-representation based design. Therefore, we were able to decrease the number of request-related messages just to **insert** and **delete**. Each of the ServerRequestDQ's sent between the client and the server begins with the **user ID** of the user who placed the request and a **request number** which is used for the identification of a request when a user receives it. Each user assigns a request number to each of his requests and increases the request counter after every one, so that each request number is unique for a given user. The main part of the message (the **body** that directly depends on an input by the user in his GUI) contains the **documentID** of the document being edited, the type of the **action** applied, its **beginning**, **end**, and the **content** of the request. Obviously, the value of end can be arbitrary for inserts, whereas the content is irrelevant for deletes. Example bodies:

- 0|INSERT|3|123|X → insert X at index 3 in document 0
    - Produces the change: ABCDE → ABCXDE
- 0|DELETE|1|3|XXX → delete segment between index 1 (inclusive) and 3 (exclusive)
    - Produces the change: ABCDE → ADE

Each request closes with a **version ID**, which has a different meaning for messages coming from the server to the client and in the opposite direction.

- For messages sent from the client to the server, the version ID is the number of the last version the client downloaded from the server → it's the version of his syncCopy.
- For messages sent from the server to the client, the version ID is simply the index of that particular change in serverHistory and therefore also the version number of the local copy the client will hold after processing the message.

Careful versioning is necessary to preserve the consistency of edits between clients. The complete grammar for a String representation of a ServerRequestDQ (which is completely interchangeable with an actual ServerRequestDQ by the use of the built-in constructor and the toString method is included in **Appendix C**:

At the end of the communication, a client sends a **bye** message which disconnects all the open sockets and stream handlers.

Thread Safety

Server:

To prevent race conditions and other concurrency-related bugs, we decided to implement an overall lock on the SEditServer object which might limit concurrency to an extent but this should not produce visible delays considering the number of users we're expecting to have connected to the server at the same time for this project. All methods in SEditServer are synchronized with the exception of serve(), which only locks on the server when it's performing an action, not when it's waiting for a new user.

Since we're intentionally exposing the rep to SEditDocument to allow for subscription management, we also need to make sure that all access from there is thread-safe. This is satisfied by design since none of the client threads access any of the SEditDocuments directly but they need to pass their messages throughs synchronized methods on the SEditServer -> all SEditDocument methods must therefore run on the main server thread.

Methods in the SEditThread class can get either get called from a SEditThread thread or from the server thread. The data in SEditThread is definitely thread safe since all the access (mainly to socket/the out object, to which the server is printing) is read only and all the fields are declared final.

Model:

Even though thread-safety should not be an issue since the GUI is designed in a completely single-threaded way in all its interaction with the model, it is implemented to satisfy all the thread-safety requirements:

All the methods in the Model classes (UserDQ, DocumentDQ, ServerRequestDQ) are synchronized when they access shared data (with the exception of applyChange which does not need to be synchronized since all its data is self-contained). This locking scheme is inefficient since it locks on the entire object for all interactions but it is the safest way to prevent concurrency bugs. In addition, in the unlikely (and unintended) event that our code should be extended to allow multiple GUIs on the same computer to interact with the same UserDQ object, the number of connected GUIs would not be enough to produce appreciable delays even using when using global locks.

GUI:

The construction of the GUI allows for a relatively simple thread safety argument. All interaction with the GUI happens within the EventDispatchThread. Every time the EventDispatchThread is used, it is with an InvokeandWait which prevents any concurrency errors. This simplicity is caused by the existence of the MessageHandlingThread which essentially acts as a buffer between server and client.

That is the entire functionality of the MHT.

Appendix A: Noteworthy Design Decisions

Pipes: We do not allow the use of Pipes in the documents or in titles. If you input a pipe to a document, a message appears warning the user that pipes are not valid characters. The carat moves to the beginning of the open document. If user attempts to name a document with a pipe, another warning appears, and the user is given another oppurtunity to name the document. This is because server messages count on pipes as special characters in their parsing

Tildas: We do not support tildas. They cannot be used in titles, and generate a pop up warning when used. The user is given another opportunity to name the document. Tildas generate new lines when typed into a text field in place of the tilda. This is becuase server messages rely on tildas to communicate newline characters.

Editing while Documents Load: You can edit a document while it is loading, despite the possibility that the user will not like the result. It is important to note that the document will still be consistent across all users, just that until the document is fully loaded the user may not have enough information to make the edits that they mean to (ie, they may delete something that would have been deleted anyways).

Opening Multiple Document Panes: The user cannot open multiple panes of the same document. A message that the document is already open will appear if the user attempts to open an already open document.

Naming of Documents: The server will not accept two documents with the same name.
If the user tries to create a doc with an already existing name, it allows the creation but renames it to the same name with an appended number (ie “document-3” rather than “document”). On the other hand, if the user tries to rename a document to a name that already exists, it is simply not allowed and the user is prompted to try another name. This is in part because we wanted users to be able to create as many unnamed documents (ie with the default name document) as they wanted before they selected a name, but once named documents are required to find a unique name.

Appendix C - Message Grammar

Maintenance messages grammar:

```
Hello ::= HELLO|UserID
Bye :: BYE
Control(Client->Server) ::= CONTROL|UserID|C2S_Tail
Control(Server->Client) ::= CONTROL|S2C_Tail
```

`C2S_Tail ::=`

- `REQUESTNEW|docName`
- `GETDOCLIST`
- `LOAD|docID`
- `CLOSE|docID`
- `RENAME|docID~docName`

`S2C_Tail ::=`

- `REQNEWPROCESSED|docID~docName`
- `DOCLIST(|docID~docName)*`
- `ERROR|messageText`
- `DOCRENAMED|docID~docName`

ServerRequestDQ messages grammar:

```
ServerRequestDQ ::= Head|Body|Tail
Head ::= UserID|RequestNumber
Body ::= DocumentID|Action|Beginning|End|Content
Tail ::= VersionID
UserID ::= a String not including "|"
RequestNumber ::= non-negative int
DocumentID ::= non-negative int
Action ::= INSERT|DELETE
Beginning ::= non-negative int
End ::= non-negative int
Content ::= a String non-including "|"
VersionID ::= non-negative int
overall:           USER_ID|REQ_NUM|DOC_ID|ACTION|BEG|END|CONTENT|
VERSION_ID
```

Appendix D: Detailed Test Strategy

Server Test Strategy:

1. one user connects to server
2. two users connect to the server
3. one user has no documents
4. one user has one document
5. one user has multiple documents
6. one user has one document with multiple inserts
7. one user has one document and does many edits
8. one user has multiple documents and does many edits
9. multiple users have one documents and do multiple interleaving edits
10. multiple users have multiple documents and do multiple interleaving edits
11. multiple users have multiple documents and rename multiple times
12. multiple users have multiple documents and rename unsuccessfully
13. multiple users have multiple documents and one user closes one document
14. invalid control request -> server ignores it
15. invalid edit request -> server ignores it

GUI Test Strategy:

Server Input Port Number Dialog

1. Input a valid, free port number -> server starts
2. Leave the port number blank -> server is not started. error dialog
3. Input an invalid port number -> server is not started. error dialog.
4. Input a port number that is already taken -> server is not started. error dialog.

Editor Input Server Address and Port Number Dialog

1. Don't start server.
    a. input server address and random port number -> error dialog
2. First start server on local computer.
    a. input "localhost" and correct port number -> connect to server. Editor window.
    b. input "localhost" and incorrect port number -> does not connect. error dialog.
    c. input blank server address (default to localhost) and correct port number -> connect to server. Editor window.

    d. input blank server address and blank port number -> does not connect. error dialog.
3. Start server on another computer.
    a. input correct server address and port number -> connects to server. Editor window.
    b. input incorrect server address and/or port number -> does not connect. error dialog.

<u>Main Editor GUI</u>

First start server and editor.

1. switch to "Open/Create" tab, double click on "There are no documents on the server" -> nothing happens
2. switch to "Open/Create" tab, click on "There are no documents on the server" and click "open" -> nothing happens
3. switch to "Open/Create" tab, enter <new document name> in the "Enter Document Name" textfield and click create -> creates the document
4. switch to "Open/Create" tab, leave the textfield blank and click "create" -> creates new document with name "Document"
5. switch to "Open/Create" tab, from the list below "Open Existing Documents", double click or click "open" on an unopened document with existing content-> opens the document in new tab and loads existing content
6. switch to "Open/Create" tab, from the list below "Open Existing Documents", double click or click "open" on an already opened -> pop up window "That document is already open!"
7. switch to one of the document tabs, click on the "x" on the tab title -> closes that document, switches back to the "Open/Create" tab
8. switch to one of the document tabs, start typing in the textfield -> text updates instantaneously
9. switch to one of the document tabs, click on "rename" -> rename dialog pops up, enter <new name>, clicks "Ok" -> dialog closes, document is renamed
10. switch to one of the document tabs, click on "rename" -> rename dialog pops up, do not enter anything, clicks "Ok" -> give notice, does not rename.
11. switch to one of the document tabs, click on "rename" -> rename dialog pops up, enter the name of an existing document, clicks "Ok" -> error dialog, does not rename.
12. Test typing functionality
    a. type regular characters (a, b, etc)
    b. enter newlines
    c. move cursor with mouse and arrow keys

d. delete characters
e. delete sections of text
f. copy paste in text
g. type pipe (should warn about typing pipes and delete it)
h. type tilda, creates new line.

Model Test Strategy

1. simpleInsertion():
    a. Description: Single user connected to ABCD, inserts X after A
    b. Desired output: User 1: AXBCD
2. concurrentInsertion()
    a. Description:
        i. Two users connected to ABCD
        ii. U1 inserts X after A, U2 inserts Y after C
        iii. Requests happen at the same time and
        iv. only get processed afterwards
    b. Desired output:
        i. Both users: AXBCYD
3. concurrentInsertionInTheSamePlace()
    a. Description:
        i. Two users connected to ABCD
        ii. U1 inserts X after A, U2 inserts Y after A
        iii. Requests happen at the same time and
        iv. only get processed afterwards, request made by
        v. user 1 gets processed first
    b. Desired output:
        i. Both users: AYXBCD
4. interleavingInsertions()
    a. Description:
        i. Two users connected to ABCD
        ii. U1 inserts X after A (1), then Y after X (2),
        iii. U2 inserts Z after C (3), then K after Z (4)
        iv. U1 places both requests 1 and 2, while U2 places 3
        v. Time t=1: Requests get processed by server in order 1, 3, 2
        vi. Time t=2: U2 receives request 1 from server
        vii. Time t=3: U2 places request 4
        viii. Time t=4: Server processes request 4
        ix. Time t=5: All remaining requests received and processed in order
    b. Ending serverHistory: r1, r3, r2, r4

    c. Desired output:
        i. Both users: AXYBCZKD
5. singleUserDeletion()
    a. Description:
        i. Single user connected to ABCD
        ii. inserts X after B -> ABXCD (INSERT X at 2)
        iii. deletes B through C -> AD (DELETE 1 through 4)
        iv. inserts Y after A -> AYD (INSERT Y at 1)
        v. gets responses for first 2 events
        vi. inserts Z after Y -> AYZD (INSERT Z at 2)
        vii. get the remaining responses
    b. Desired output:
        i. User: AYZD
6. tripleUserDeletion()
    a. Three users connected to ABCDE
        i. R1: user 1 deletes BC -> sees ADE (DELETE 1, 3)
        ii. R2: user 2 deletes CD -> sees ABE (DELETE 2, 4)
        iii. R3: user 3 deletes BCD -> sees AE (DELETE 1, 4)
    b. Requests get processed in order
    c. User views:
        i. U1: R1->ADE, R2->AE, R3->AE
        ii. U2: R1->AE, R2->AE, R3->AE
        iii. U3: R1->AE, R2->AE, R3->AE
7. multipleDocsTest()
    a. Two users connected to two document, the documents shouldn't affect each other.
        i. ABCD gets loaded into document 0
        ii. User2 inserts Y into document 1
        iii. User1 inserts X after A in document 0
    b. Requests get processed and displayed
    c. Both users hold AXBCD in document 0 and Y in the document 1

The point of the test cases described above is to push the concurrency capabilities of the implementation to the limits with many changes happening at the same time without knowing about other changes made by other users.

As shown above, the current implementation can handle extreme lags when one client only gets the information about the document changing when other clients made many mutually interfering changes.

System Test Strategy:

Note that tests are broken into sequences, where the test relies on the actions that precede it in the sequence. ==== indicates a test. ====

Sequence 1: Renaming and Document List Updates

1) Start the server, open User0.
2) Create a document "doc1" from User0
3) open User1.
4) ==== check User1 sees doc1 in loaded docs====
5) open User2 first, then create new doc "doc2" from User1
6) ==== check User2 sees doc2 in their doclist ====
7) move User0 to Open/Create Panel,
8) ==== check "doc2" is also visible from User0 (not initially in Open/Create) ====
9) move User1 to Open/Create Panel.
10) open doc1 in User2.
12) move User0 to doc1. Rename doc1 to "renamedDoc1"
12) ==== check that User1 seed the name change in their docList====
13) ==== check User2 sees the name change in their open doc1 ====

Sequence 2: Saving document changes

1) Start the server, open User0
2) Create a document "doc1" from User0
3) add the message hello to doc1
4) open a User1, open doc1
5) ====check that doc contains "hello"====
6) add the line "what's up" to doc1 from User1.
7) ==== check that it appears for User0 ====
8) Close doc1 from both User0 and User1.
9) Open doc1 from User0
10) === check that doc1 contains "hello what's up" ====

Sequence 3: Simultaneous Edits

1) Start server, open 3 users, open to same document.
2) All go to own line, begin typing
3) ==== check that it displays properly ====
4) All type on SAME line
5) ====check that displays what is typed properly, same copy, and carat

updates well ====

6) One user highlights a section of the text, other users type, copy-paste, delete, block-delete INSIDE the selection
7) ==== check that selection updates to include the new text ====
8) Delete a selection which includes the carats of the other users
9) ====check that carats are moved to beginning of deleted area ====
10) One user highlights a selection, other user deletes the entirety of the selection
11) ====check carat is moved to beginning of deleted area ====
12) A user enters a pipe,
13) ==== error message appears warning the user, pipe disappears, carat is moved to the beginning of the text pane ====

Stress Tests-- successful is nothing breaks

1) Have a number of users (>3) connect to the same document, type as fast as humanly possible (multiple keyboards encouraged)
2) Have many users connect to multiple documents, again type a bunch of things
3) Open many many documents, load many many tabs.
4) Type in Czech, or chinese. will display ascii equivalents.

Appendix E: Description of the Classes

The purpose of every class is described here. Methods and fields are all listed but only described *here* if they are deemed either important or complicated enough to merit a description. Getters and setters in particular will be ignored.

Package: **Server**

SEditServer:

Description: The SEditServer class is the main entry point of the server side of the application. It sets up the server, listens for incoming client connections, and manages the client-document communication

Fields:

Map<String, SEditThread> users: Maps userIDs to SEditThreads so that messages can easily sent to a thread based on userID
int userIDCounter: Number of users that have connected
Map<Integer, SEditDocument> documents: Maps documentIDs to the actual document Objects.
boolean die: Set to true on close to signal a proper close
int PORT_NUMBER:
ServerSocket serverSocket:
int documentIDCounter:

Methods:

public SEditServer(int port): Creates a new SEditServer object listening for connections on the specified port
public synchronized void handleRequest(String input): Reacts appropriately to an incoming request:
Incoming CONTROL requests:
-- CONTROL|<userID>|<requestType>|<message dependent tail>
--- GETDOCLIST, tail: empty
--- REQUESTNEW, tail: suggested document name
--- RENAME, tail: documentID~suggested name
--- LOAD, tail: documentID
--- CLOSE, tail: documentID
Outgoing CONTROL messages:
- CONTROL|<messageType>|<message dependent tail>
--- DOCLIST, tail: (|documentID~documentName)
--- REQNEWPROCESSED, tail: |documentID~documentName
RequestServerDQ requests are passed on to their respective target documents
private synchronized void distributeMessage(String message):Sends a message to all the users connected to the server
private synchronized String getDocListMessage(): Generates a DOCLIST message to be sent to the users
public synchronized void removeUser(SEditThread user): Unsubscribes the user from all the documents he is registered for and removes him from the server

public void serve(): Listens at the serverSocket for incoming connections and creates a SEditThread for them when it hears one
public synchronized void kill(): Closes the serverSocket and shuts down the server
public static void main(String[] args):The main method that creates and launches the server

SEditDocument:

Description: SEditDocument represents the server version of a document. It is recognized by an immutable documentID and a changeable documentName. The document itself is stored as a sequence of changes to the initial empty document.

Fields:

List<ServerRequestDQ> serverHistory: Contains every ServerRequestDQ that the document has processed.
List<SEditThread> subscribedUsers: A list of every user receiving updates on this document
int documentID:
int documentName:

Methods:

public SEditDocument(int documentID, String documentName): Creates a new SEditDocument with the given ID and Name
public void processRequest(String input): Processes a request sent in by a user, makes it match with current server indices and distributes it to the subscribed users
public void distributeMessage(String message): Registers a user with this document so that he received all future updates related to it. A user cannot be added multiple times.
private String anonymizeMessage(String message): Removes the user name from a message to make sure loading is processed properly
public void rename(String newName):
public void unsubscribeUser(SEditThread user):
public void subscribeUser(SEditThread user):
public int getDocumentID():
public String getDocumentName():

SEditThread:

Description: The SEditThread class represents a connection to an individual client. It listens to messages from users and redirects them to the server queue. The server also uses its sendMessage() method to inform users about updates made by other users.

Fields:

BufferedReader in: BufferedReader that reads in from its user
PrintWriter out: PrintWriter that prints to the SEditThreads user.
Socket socket:

SEditSever server;
String userID:

Methods:

public SEditThread(Socket socket, SEditServer server, String userID): Creates a new SEditThread object to communicate with a client on the specified socket
private void handleConnection(Socket socket): Listens to incoming messages from the client and redirects them to their destination
public void sendMessage(String message): Sends the specified message to the user
public void run():
public String getUserID():

Package: **Model**

DocumentDQ:

Description: DocumentDQ represents the client side of a document. It stores the most current synchronized version and a queue of local updates to be applied to it before it is displayed to the user.

Fields:

String lastMessageReceived: The most recent message received from the server
List<ServerRequestDQ> localQueue: A list of messages making up the localQueue, which is updates that have not been applied to the server yet, but have been applied to the local copy.
int syncVersion: versionID that document is on.
String syncCopy: internal copy of the String representing the document
String userID:

Methods:

public DocumentDQ(String userID): Creates a new instance of DocumentDQ and saves the owner ID into a private field
public void pushRequest(String requestText): Accepts a request that's targeted towards this document and processes it
public void addRequest(String request): Adds a request to the local queue to be displayed to the user immediately after inputting it
public String applyChange(String source, ServerRequestDQ request): Applies the specified edit request to a String and returns the new version
public String updateSelection(String requestText): Takes in a dummy request produced by the GUI to represent the current selection and applies the last request received from the server to it to get a new selection

public String getView():
public String getSyncCopy():
public int getCurrentVersion():

ServerRequestDQ:

Description: ServerRequestDQ represents an insert/delete request sent to or received from the server. String->ServerRequestDQ is provided in addition to a symmetrical toString() method to ensure complete interchangeability of a String representation following the specified grammar and an instance of ServerRequestDQ.

Required request format:
USER|REQ_NUM|DOC_ID|ACTION|BEG|END|CONTENT|VERSION_ID

This class also manages request-request interference and versioning to ensure proper synchronization.

Fields:

String action: String that says what the action the request asking for
String content: The data for the update
String userID:
int documentID:
int requestNumber:
int versionID:
int beginning:
int end:

Methods:

public ServerRequestDQ (String requestText): Parses a server request passed in in its String representation and creates a new ServerRequestDQ object
public ServerRequestDQ(ServerRequestDQ request, int newVersionID): Copies an existing request but changes the versionID
public void applyUpdate(ServerRequestDQ previous): Applies an update that's received after the current one but which actually precedes it in the logic of the server updates the current update to match the version produced by the previous update. E.g. Someone else submitted an update that inserts three characters at index 3 and concurrently I made an update that inserts a character at index 15. Since their update was processed first by the server, I want to "update" mine by increasing its index by 3 as if I made it to the more recent version.
public String getAction():
public int getBeginning():
public int getEnd():
public String getContent():
public int getDocumentID():
public int getVersionID():
public String getUserID():

public int getRequestNumber():
public void setUserID(String newUserID):
public String toString():

UserDQ:

Description: UserDQ is the main class in the model package, iIt manages the current versions of the documents and their updating on the client side.

Fields:

Map<Integer, DocumentDQ> documents: A mapping from document IDs to the actual document objects associated with them.
int requestCounter: number of requests that have been sent by the user.
List<String> serverBuffer: buffer/queue for newly created requests before being sent to server.
String userID:

Methods:

public UserDQ(String userID): Creates a new UserDQ object with the specified userID
public synchronized void createRequest(int documentID, String requestBody): Creates a request to be sent to the server based on user's interaction with the GUI For testing, this simulates event:
Message sent by the user:
request: USER|REQ_NUM|DOC_ID|ACTION|BEG|END|CONTENT|VERSION_ID
request body: ACTION|BEG|END|CONTENT
public synchronized void addDocument(int documentID): Registers a document with the UserDQ object so that all incoming requests can be properly redirected
public synchronized void pushRequest(String requestText): Forwards a request coming in from the server to its target document For testing, this simulates event: message received by client after being sent by the server.
public synchronized String pullRequest(): Pulls a message to be sent to the server next from the serverBuffer. For testing, this simulates event: Request received by server
public synchronized String updateSelection(String requestText): Forwards the updateSelection requests to the document they are related to

public synchronized String getView(int documentID):
public synchronized String getSyncCopy(int documentID):
public synchronized String getUserID():

Package: **View**

DocPanel:

Description: Subtype of JPanel for viewing and editing a document. Displays a text field
and rename button.

Fields:

DocumentListener listener: DocumentListener that is listening for text pane changes
JTextPane text: Text pane where the document is displayed and changes are made
String docName:
int docNum:
long serialVersionUID:

Methods:

public DocPanel(final int docNum, String docName, final Editor editor): Creates a new DocPanel with a documentId, document name, a document pane, a listener for the TextPane.
public void paintComponent(Graphics g): paints the panel like a regular Jpanel
public void repaint(): repaints the panel like a regular Jpanel
public JTextPane getTextPane():
public int getNum():
public String getName():
public void setName(String newName):

Editor:

Description: Editor is the main GUI class. Represents the main communication device between the server and the user. It contains a tabbedPane which contains all of the documents and document loading panes for a given user.

Fields:

JTabbedPane tabbedPane: JTabbedPane where all document and load document Panels are stored
UserDQ user: user instance associated with this GUI

int ignoreNext: flag for indicating whether the GUI updates should be sent to server
MessageHandlingThread MHT: instance of MHT that passes server messages to this GUI
HashMap<Integer, DocPanel> docIDtoDocPanel:
Socket serverSocket:
PrintWriter out:
BufferedReader in:
long serialVersionUID:

Methods:

public Editor(): Creates new Editor object with a tabbed pane, opens a message handling thread. Opens an open document pane.
public void initTabComponent(int i): makes a close button on the tab.
private void handleServerGreeting(): reads the first message from the server, which assigns a userID. shows an error if the server doesn't respond properly.
public void sendMessage(String message): sends a message to the server.
public void handleLine(String line): parses server messages. If NOT a control message, passes the message to the user and updates the TextPane. If a control message, applies the update to the Editor.
public void updateView(String docID): updates the textPane of the panel corresponding to the give docID to add the new updates from the server.
public String createControlMessage(String messageType, int docID, String docName): Create a new message from the given parameters that can be understood by the server.
public static void main(final String[] args): Runs the editor. Sets the dimensions etc. If server cannot be reached or sends a bad message, creates a pop up to tell the user there was an error.
public void showGreetingDialog(): Shows the greeting dialog and prompts the user for the server address and port number. Handles user input, and connects to server if input is correct.
public UserDQ getUser():
public void setUser(UserDQ user):
public Socket getServerSocket():
public void setServerSocket(Socket serverSocket):
public int getIgnoreNext():
public void setIgnoreNext(int ignoreNext):
public JTabbedPane getTabbedPane():
public void setTabbedPane(JTabbedPane tabbedPane):
public HashMap<Integer, DocPanel> getDocIDtoDocPanel():

public void setDocIDtoDocPanel(HashMap<Integer, DocPanel> docIDtoDocPanel):

HintTextField:

Description: The custom subclass of JTextField, which handles the "hint" part of the "Create New Document" section in DocumentSelectionPanel.

Fields:

Font gainFont: “regular” font

Font lostFont: italicized font

Methods:

public HintTextField(final String hint): Constructs a hint text field, which is used for creating a new document and adding the title.

NewDocPanel:

Description: Essentially a placeholder JPanel while the documents load from the server. Tells the user that the document list is loading, and the server that the document list should be sent. Usually does not display for long enough to be noticed.

Fields:

long serialVersionUID:

Methods:

public NewDocPanel(final Editor editor): Creates a NewDocPanel to tell the user documents are loading. Sets the minimum size similar the window size. tells the editor to ask for the document list.

public void paintComponent(Graphics g): paints the panel like a regular Jpanel

public void repaint(): repaints the panel like a regular Jpanel

DocumentSelectionPanel:

Description: DocumentSelectionPanel is a class to represent the document selection menu. It allows the user to either load a document from the server or create a new document.

Fields:

JList docsList: JList which holds DocumentIDsAndNames object to display the documents on the server

JTextField docNameTextField: field for new document names

JScrollPane scrollPane: allows docsList to be scrollable

JButton createButton: create new button

JButton openButton: open from server button

JLabel openLabel:
JLabel createLabel
JLabel orLabel:
long serialVersionUID:

Methods:

public DocumentSelectionPanel(String[] list, final Editor editor): creates a new DocumentSelectionPanel. Will display the list of document from the server that can be selected on one side (or a message that there are no documents on the server). On the other side there will be a text field to create a new document with the given name (or a variation of the name if its already taken, ie. Niki-3 is Niki is already taken).
private void openDocument(Editor editor): Opens a new panel with the selected value from the list if it is not already open on the client. Otherwise tells the user that the document is already open. Sets focus on the new Pane. sends Load request to the server for the newly opened document.
public void paintComponent(Graphics g): paints the panel like a regular Jpanel
public void repaint(): repaints the panel like a regular Jpanel

Listeners:

On **docNameTextField:** sends a new doc request on enter from the text field
On **createButton:** sends a new doc request button click with name from text field
On **docsList:** sends a load doc request on double click from the list. Also opens the new tab after checking that the document is not already open on the client.
On **openButton:** sends a load doc request on button click. Also opens the new tab after checking that the document is not already open on the client.

DocumentIDsAndName (contained in DocumentSelectionPanel):

Description: Wrapper class for displaying elements in the JList.

Fields:

int num: docID of represented doc
String name: name of represented doc

Methods:

public DocumentIDsAndNames(String ob): Creates new object based on passed in doc identity string
public int getNum():
public String toString():

ButtonTabComponent:

Description: Component to be used as tabComponent; Contains a JLabel to show the text and a JButton to close the tab it belongs to

Fields:

JtabbedPane pane: pane to which ButtontabComponent belongs

Editor editor: editor to which ButtontabComponent belongs
long serialVersionUID:
MouseListener buttonMouseListener: Listens for mouse events on the button

Methods:

public ButtonTabComponent(final JTabbedPane pane, Editor editor): Constructor. creates the appropriate ButtonTabComponent and sets the GUI.

TabButton:

Description: Custom button for the close button

Fields:

long serialVersionUID:

Methods:

public TabButton(): Constructor. sets appripriate GUI elements.
public void actionPerformed(ActionEvent e): handles when the user clicks the "x" icon on the tab, removes the tab and sends a message to the server to unsubscribe for messages about this document.
protected void paintComponent(Graphics g): paints the "x" component for handling close tabs
public void updateUI():

Package: **Controller**

DocumentContentListener:

Description: DocumentContentListener is a custom Document listener that waits for updates in the editor and creates the messages to be sent to the server.

Fields:

int docNum:
Editor editor:

Methods:

public DocumentContentListener(int docNum, Editor editor): Creates new DocumentContentListener on passed in document
public void update(DocumentEvent e, EditType action): update method for handling user inserts and deletes in the Editor to the document. Gets the correct locations and sends the appropriate messages to the server.
public void insertUpdate(DocumentEvent e):
public void removeUpdate(DocumentEvent e):
public void changedUpdate(DocumentEvent e):

MessageHandlingThread:

Description: The Message Handling Thread's purpose is to listen for messages from the server and pass them into the editor's message handler. It uses invoke and wait to prevent concurrency issues. There is just one MessageHandlingThread per editor, and it shares a BufferedReader with the editor.

Fields:

Editor editor: editor to which the thread passes messages

BufferedReader in: reader from which the thread recieves messages.

Methods:

public MessageHandlingThread(Editor editor, BufferedReader in): Constructor for the MessageHandlingThread. Connects the thread to its editor and reader.

public void handleConnection(): Handles the incoming messages from the server.Reads in lines, closes the reader on a "BYE" message, and passes all other messages to another method to be sent to the editor.

public void sendToEditor(final String line): Passes the passed in message to the editor's handleLine method. Uses an invoke and wait and a runnable to ensure that only one message is sent at a time.

public void run(): When run, thread attempts to handle the connection.

Appendix F: History

We initially planned to use a double-linked-list-based representation for the document with each of the nodes containing an element of the text (the size of which could depend on the implementation but would most likely be a single character). Each of the nodes would have been identified by its unique ID which would prevent any confusion about indices as other users' changes modify the text. This way, we could achieve true concurrency as multiple nodes could be modified at the exact same time without interfering with each other. Minor issues related to this system (such as finding the ID's of the elements changed in the GUI where we only see a String representation of the linked-list) would have been possible to resolve and we drafted some of the proposals along with the entire initial design in Appendix G.

As the number of issues related to this more robust but much more complex implementations kept increasing (the number of messages a server processing the necessary requests grew, desired behavior required the addition of many ad hoc work-arounds etc.), we decided to recreate the initial design concentrating on simplicity as opposed to a design that's elegant as an idea but more complicated in implementation.

The new design removes true concurrency and implements a system of 2 queues, one held by the server and one by each of the users. The main points of this design are:

1. The changes made by a user in their GUI are immediately reflected in what they see but their local copy of the document is not directly mutated
2. The local copy held by a user is just a simple String, not a linked list
3. Only the server has the capability to mutate individual users' local copies of the document.
4. The server keeps a shared queue of all the changes made by all the users in the order they are
5. received. Along with point 3, this guarantees that at the end of the day, all users must necessarily hold the same local copy of the document, regardless of the order of changes they see in their GUI.
6. The actual content of the document is never directly distributed and the server does not directly store it. Each document is represented as a series of changes (i.e. differences from the previous version) that when applied in order can recreate the document at any state since it's creation.

Appendix G: Rejected Initial Design

GUI:
Built with the Java Swing library. Each user, when connected to the server, will be updated with all the documents currently stored in the server. When each user edits the document, their local changes will be displayed instantaneously on their own GUI. These changes will also be submitted to the server into the server's "edit queue." Once an item on the "edit queue" is processed by the server, the server sends an updated message to each client using a protocol described in the Communication Protocol section. The GUI contains: 1) a "New Document" button that creates a new document, 2) a "Change Name" button that renames the document, 3) several tabs, each containing a document, and 4) the text pane that contains the body of the document.

Backend:
Server: The server performs the following tasks: 1) maintains a queue of edits named "edit queue", 2) processes each client's edit request, placing them in order of the server receiving the edit requests from the clients, determines whether these requests are still valid, and accept valid requests as appropriate, 3) every time a valid request is accepted, sends an update request to the client, 4) keeps an auto-created copy of a client which does not post any update requests and receives all updates from other clients. NOTE: The server's main job is to maintain and process the edit queue. It does not keep a "server" copy of the documents, instead, it keeps a copy of a "client" to back up changes.

Client: Each client 1) keeps a copy of the documents, 2) sends and receives update requests to/from the server.

Classes: The Model part of the Collaborative Editor has the following classes: 1) Anchor, 2) Document, 3) Node.

Edit: An edit by a user is defined to be one of the following: 1) insert, 2) delete, 3) block insert, 4) block delete.

Anchors: The server will store an "anchor" for each user. This is essentially just a pair of unique id's which indicate where a user's cursor is in the document. The first id corresponds to the node before the user's cursor, while the second id is the node after the cursor. The reason that both are stored is to handle simultaneous insertions by users. If the server sees that a user's anchor has nodes that are not consecutive, then it can tell that two user's have been editing in the same location and will send messages back to the clients appropriately. The details are described in the Insertion method description

Documents: Each Document contains a String representing the Document Name, an int representing a unique Document ID, a LinkedList of Nodes representing the content of the each document, and an ArrayList of Anchors that holds all the Anchors for all the users.

Nodes: A node is an immutable object that holds the base units (characters, punctuation, white space, tab, newline character, etc) that make up the content of the Document. Each nodes has an int representing a unique ID the server assigns to each base unit and a String representing the actual base units.

Client Actions/Edits:

Insertions:

- **Single character insertions**: To handle insertions, we are assuming that all the users are holding valid anchors which specify between which 2 nodes their cursor is. Entering a character will then result in sending an "insert CHARACTER" message to the server - the message from the user does not (and should not) specify the actual index at which they are inserting, that would cause problems in concurrent editing since indices change based on other users' actions. The server then evaluates whether the anchor has been "invalidated" by previous insertions (it no longer connects 2 consecutive nodes) in which case it has to be updated to produce the desired behavior as specified by the documentation. If the anchor is still valid, a new node is inserted and the position of the anchor is shifted to point to it.
- **Block insertions (copy-and-paste)**: In the server queue, a block insertion is just a sequence of single-character insertions. It cannot be sent as such from the client though since that would result in possible interleaving with other users' requests. The client will therefore be able to send another type of message to the server that groups these insertions together as an atomic request. The server then decomposes it for the queue and distributes it to all the clients as simple insertions.

Deletions:

- **Single character deletions**: After the user has acquired an anchor in the text, pressing backspace / delete results in sending a message to the server with a deletion request for the immediately preceding / following node. The requests contain the unique ID of the node to be deleted. In that way, simultaneous deletion requests for the same node do not result in more nodes getting deleted than intended. (We rejected the idea that a person just sends a delete request relative to their anchor as those would produce undesirable behavior). All anchors connected to the deleted node will be shifted to the node immediately before.

- **Block deletions**: This is very similar to single character deletions except both the ID of the first and the last node to be deleted have to be sent to the server by the client. The server then loops through the LinkedList and shifts all the anchors connected to any of the nodes to the node that immediately precedes the beginning of the deleted block.

Protocols:

Most messages to the server will be structured in 3 parts.
1. “who” will be a string that identifies which user is sending the message to the server. For example, “u2” would indicate that user 2 is sending the message.
2. “what” will indicate which of the actions is being passed. The options are *insert, delete, block-delete block-insert, update-anchor.*
3. “content” will contain one of a few things depending on the “what”. If it is insert, it will contain the character to insert. A block insert will contain the string to insert. Single deletions will have the unique id being removed, block-deletions will have the start and end ids of the block being deleted. If it is an update anchor request, then it would contain the unique ids of the “start” and “end” locations of the anchor update.

There are a few “special” messages. new doc, update document name, and bye which are only sent from the client to the server unlike the other messages.